VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Porovnání implementace garbage collectorů různých programovacích jazyků

Comparison of implementations of garbage collectors of different programming languages

2009 Vlastimil Čadra

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 21. srpna 2009 …….

Rád bych zde poděkoval panu inženýru Zdeňku Sawovi za to, že mě vedl a poskytl mi důležité literární zdroje.

Abstrakt

Tato práce pojednává o implementacích garbage collectorů v jazycích Python, Objective Caml a Scheme. Nezabývá se, ale pouze samotnou implementací, ale také teoretickým popisem použitých algoritmů. Snaží se popsat tyto implementace nejen informacemi, které byly vyčteny ze zdrojových kódů a dokumentací, ale i vzájemným porovnáním obou garbage collectorů. Cílem přitom není určit, který z nich je lepší, ale pouze popsat jejich vlastnosti, říct ve kterých svých vlastnostech se liší a které mají naopak společné. Popsat průběh jejich práce. K tomu je také nutné popsat, alespoň trochu, strukturu objektů, které jsou alokovány na haldu.

Klíčová slova

garbage collector, Mark-and-Sweep, počítání referencí, kopírovací algoritmus, generační garbage collector, inkrementační algoritmus , OCaml, Python, minoritní halda, majoritní halda, Scheme, Boehmův algoritmus

Abstract

This thesis is about implementation of garbage collectors in lanuages of Python, Objective Caml and Scheme. It is not only about implementation itself but it also about theoretically description of used algorithms. The thesis try to describe these implentations not only by an informations, which I read from source code, but also by mutual compare the garbage collectors. It's not goal of the thesis to specify, which of the garbage collectors are better, but only describe their properties and tell which their properties are different and which are same. Goal is describe their work flow. It's necessary for that to describe, just a little, structure of their objects, which are allocated on heap.

Key words

garbage collector, Mark-and-Sweep, reference counting, copying algorithm, generation garbage collector, incremental algorithm, OCaml, Python, minor heap, major heap, Scheme, Boehm`s algorithm

Obsah

1. Úvod 4
2. Obecné algoritmy používané pro garbage collecting 5
   2.1. O paměti 5
   2.2. Metoda počítání referencí 7
   2.3. Mark-and-Sweep algoritmus 8
   2.4. Kopírovací algoritmus 10
   2.5. Generační garbage collector 11
   2.6. Inkrementální algoritmus 12
   2.7. Boehmův algoritmus 13
3. OCaml 15
   3.1. Popis struktury paměti jazyka Ocaml 15
   3.2. Popis funkce garbage collectoru v jazyce OCaml 17
4. Python 21
   4.1. Stručný popis objektů 21
   4.2. Popis činnosti garbage collectoru 22
5. Scheme 26
   5.1. O objektech na haldě 26
   5.2. Popis činnosti garbagr collectoru 26
6. Porovnání implementací garbage collectorů 30
7. Závěr 32
8. Literatura 33

1. Úvod

Tato práce se zabývá analýzou implementací garbage collectorů v konkrétních implementacích programovacích jazyků.

Technologie garbage collectorů je stará více než čtyřicet let a jedná se o výraznou pomoc pro programátory při implementaci aplikací. Zatímco u jazyků, jako je třeba C/C++, je správa paměti zcela v rukou programátora, což od něj vyžaduje značnou kázeň a práci navíc, tak u jazyků jako je na přiklad JAVA se ke správě paměti využívá garbage collector, který celý proces odstraňování objektů z paměti automatizuje a programátor se tak nemusí téměř o nic starat.

Tato výhoda ale není zadarmo. Využití garbage collectoru může výrazně zpomalit běh aplikace. Navíc ty jednodušší algoritmy, jako je např. počítání referencí, trpí různými nedostatky jako je například neschopnost odstraňovat zacyklené objekty. Pokročilejší algoritmy, jako je Mark-and-Sweep, které odstranili nedostatky těch jednoduchých, běh aplikace dokonce na chvíli přeruší, což může být pro některé systémy nepřípustné a i tyto algoritmy mají své nedostatky. Ty nejpokročilejší techniky pro garbage collecting se sice dokáží vypořádat s nedostatky svých jednodušších kolegů, ale jedná se o velmi složité implementace využívající většinou kombinaci několika algoritmů pro odstraňování mrtvých objektů z paměti.

Proto stále ještě existují jazyky, a jedná se většinou o jazyky pracující na nižších úrovních, tedy blíže procesoru, kde potřebujeme co nejrychlejší zpracování, které garbage collector ve svých implementacích nevyužívají.

V druhé kapitole se zabývám obecným teoretickým popisem principu jednotlivých algoritmů, metod a přístupů , které slouží k vyhledávání a uvolňování objektů, které již nejsou dostupné a v paměti tak pouze zabírají místo. Právě tyto algoritmy se používají k implementaci garbage collcetorů. Na samém začátku kapitoly také rozebírám principy samotné alokace objektů na hladu a správu paměti jako takovou.

V třetí kapitole je pak popsána práce garbage collectoru v implementaci jazyka OCaml. Je zde popsán průběh minoritní i majoritní kolekce včetně případné defragmentace paměti. Rovněž je zde popsána i struktura obou hald a jednotlivých objektů, které lze na tyto hlady alokovat.

Ve čtvrté kapitole je popsán garbage collector, který je použit v implementaci jazyka Python a to včetně popisu činnosti detektoru cyklů, který má za úkol vyhledávat a rozbíjet cykly objektů, protože běžné počítání referencí, které je zde pro uvolňování paměti využito si s tím neumí poradit.

V páté kapitole je popsána implementace Boehmova algoritmu v jedné z implementací jazyka Scheme. Jsou zde popsány jednotlivé fáze v nichž se uvolňování objektů odehrává.

V šesté kapitole jsou pak jednotlivé implementace porovnány dle použitých algoritmů a svých vlastností jako je třeba fragmentace paměti či rozeznávání silných a slabých referencí.

2. Obecné algoritmy používané pro garbage collecting

Tato kapitola popisuje obecné metody a algoritmy, které se používají pro implementaci garbage collectingu. Kapitola je založena na informacích získaných knihy Modern Compiler Implementation in C [1]. Všechny algoritmy, popsané v této kapitole jsem převzal z výše zmíněné knihy.

2.1. O paměti

Paměť programu tvoří statická a dynamická data. Ta statická představují záznamy o globálních a lokálních proměnných programu a je jim vymezen pevně daný kus paměti, který se za celou dobu běhu programu nezmění, i když může někdy být alokován s určitou tolerancí. Kdyby program obsahoval pouze statická data, nepotřebovali bychom žádnou dynamickou správu paměti a tedy ani garbage collectory.

Jenže jsou zde ještě dynamická data, u kterých není možné předem určit jakou budou mít velikost, typ či jak dlouho budou tzv. živé, což znamená jak dlouho na ně budou existovat odkazy se statických dat.

Těmto datům se říká objekty a alokují se do dynamické části paměti, které se říká halda. Na rozdíl od statické části paměti nemá halda pevnou délku a je zcela běžné, že během běhu programu dochází k jejímu rozšiřování a někdy i k zmenšování. Halda má většinou délku rovnou celým násobkům velikosti stránky virtuální paměti a je tvořena seznamem bloků. Ty tvoří souvislý kus paměti a jsou očíslovány (nejedná se o nic jiného než o adresy v paměti), aby mohli být od sebe rozpoznány, a aby mezi nimi mohlo být iterováno.

Objekty na haldě jsou spolu spojeny referencemi, které nám říkají, které objety spolu komunikují přímo, a které pomocí jiných objektů. Pokud od nějakého objektu vede reference k jinému objektu a je-li onen první objekt dostupný, pak je dostupný i objekt, na který odkazuje. Platí též, že pokud objekt vytvoří jiný objekt, vede od rodiče k potomkovi reference.

Díky výše popsanému, jsi lze představovat hladu jako orientovaný graf, kde objekty tvoří uzly grafu a reference mezi objekty jsou pak orientovanými hranami. Mnoho metod garbage collectingu proto využívá k vyhledání nedostupných objektů na haldě metody, které se využívají k procházení grafu.

Obecný popis alokace

Teoreticky lze dynamické objekty alokovat téměř do nekonečna, protože jakmile nám dojde paměť, může program požádat operační systém o novou. Díky tomu se sice nebudeme muset starat o správu paměti a problémy s tím související, ale na druhou stranu riskujeme vážné zatížení procesoru a paměti, které navíc není nekonečno a jednou dojít prostě musí. Tento způsob je totiž absolutním plýtváním paměti, jak bylo zjištěno, objekty alokované v paměti nežijí až na výjimky dlouho, takže v tomto případě bychom, pokud vezmeme v úvahu nejhorší scénář, měly paměť počítače, zcela zahlcenou daty, které nejsou k ničemu, protože objekty v paměti, které tyto data obsahují jsou téměř všechny mrtvé. Takový program by navíc znemožnil nebo alespoň vážně zpomalil běh ostatních aplikací, protože by paměť, kterou by mohly použít pro svůj běh, nenasytně zabíral pro sebe. Samozřejmě lze na druhou stranu tuto metodu použít u programů, jejichž nároky na paměť jsou velmi malé a jimi alokované objekty žijí dlouho.

Nejjednodušším způsobem alokace objektů na haldu je pouhé posouvání ukazatele na konec naalokovaného prostoru. Celé to vypadá tak, že na začátku ukazuje náš ukazatel na začátek haldy, v momentě kdy chceme alokovat objekt pak tento ukazatel posuneme o velikost objektu, takže nyní ukazuje na konec právě naalokovaného objektu a tedy i do prostoru kam budeme v budoucnu alokovat nový objekt. Jakmile budeme alokovat další objekt, znovu posuneme ukazatel o jeho velikost. Tento

způsob se skvěle hodí k metodě, kdy alokujeme objekty bez toho, abychom se starali o jejich odstranění, protože se zde vůbec nepředpokládá, že bychom odstranili objekt, který se nachází, někde uprostřed haldy. Kdybychom to totiž učinili, tato metoda nám neposkytuje žádné prostředky, jak se k tomuto uvolněnému místu vrátit a alokovat na něj nový objekt. Proto zde buď nemůžeme odstranit objekty vůbec a nebo je odstraníme všechny naráz a potom přesuneme ukazatel zpátky na začátek haldy (jak se to dělá např. u kopírovacího algoritmu).

Pokud chceme mít možnost např. odstranit objekt v okamžiku jeho smrti (tedy v momentě kdy se stal nedostupným), musíme být zároveň schopni se na jeho pozici vrátit a alokovat zde nový objekt či objekty. To nám umožní využití technologie *freelistů*. Freelist není nic jiného než seznam adres objektů, které byly odstraněny z paměti a představují volné místo. Chceme-li nyní alokovat nový objekt, musíme se podívat na freelist, zjistit zda se tam nachází blok paměti odpovídající velikosti, přejít na jeho pozici v paměti a vytvořit zde nový objekt. Celý proces je tedy mnohem složitější než předchozí přístup, nicméně nám dovoluje paměť dynamicky spravovat. Nyní můžeme odstranit libovolný objekt na haldě a na jeho místo následně alokovat objekt nový. Při odstranění objektu se jeho adresa uloží do freelistu a my tak víme, kde přesně se v paměti tento objekt nacházel a můžeme tak jeho místo znovu využít.

Freelisty mají dvě výrazné slabiny. Tou první je značné prodloužení doby nutné k alokování objektu. Zatímco předtím nám jen stačilo alokovat objekt tam, kde byl ukazatel a ten následně posunout, zde u freelistů musíme nejprve najít vhodné místo, kde můžeme objekt umístit. A v tom je právě problém. Freelist je seznam a nám nezbývá nic jiného, než jej sekvenčně projít, abychom v něm nalezli vhodný záznam, pak je ale složitost takového vyhledávání rovna *O(n)*, kde *n* je počet záznamů ve freelistu. Určitým východiskem by samozřejmě bylo udržovat freelist setříděný, pak by byla složitost $O(\log_2 n)$, protože bychom mohli použít metodu půlení intervalu, nicméně bychom zase spotřebovali spoustu času při odstraňování objektů, protože bychom i při odstranění jediného objektu museli freelist setřídit. Proto se používá tzv. *pole freelistů*. Jeho základem je, že udržujeme skupinu freelistů z nichž každý obsahuje pouze bloky konkrétní velikosti (nemáme ale freelist pro každou velikost, to by bylo příliš nákladné), při alokaci tak jednoduše můžeme odebrat první záznam z freelistu, který spravuje bloky o velikosti jenž nám vyhovuje nejlépe. Pokud je takový freelist prázdný, vybereme záznam s nejbližšího vyššího freelistu, který není prázdný a objekt vytvoříme běžnou cestou s tím, že nadbytečné místo zůstane k objektu "přilepené" dokud nebude odstraněn.

Tím se vlastně dostáváme k druhé slabině freelistů. Schopnost odstranit objekty odkudkoliv z paměti a následně na jejich místo alokovat objekt nový, sebou totiž přirozeně přináší nevýhody *fragmentace*. Známe dva druhy fragmentací:

1. **Externí fragmentace** – Nastává pokud při alokování objektu o velikosti *n* máme sice k dispozici spoustu volných bloků, ale všechny jsou menší než *n*. Ačkoliv jejich celková velikost je větší
2. **Interní fragmentace** – Nastává pokud alokujeme objekt do bloku o větší velikosti, aniž bychom tento blok rozdělili. V takovém případě daný objekt zabírá místo bloku do nějž byl alokován místo toho, aby zabíral pouze svou velikost, která je menší. Navíc je pro nás oblast paměti, která tvoří rozdíl mezi velkostí alokovaného objektu a velikostí původního bloku, nedostupná do té doby, dokud nebude objekt z paměti odstraněn.

Je proto nutné, alespoň čas od času hladu na níž se využívá freelistů defragmentovat.

Manuální a automatická správa paměti

Jak již bylo zmíněno v úvodu, existují dva způsoby jak spravovat paměť programu. Tím prvním je manuální správa paměti, kdy programátor řídí alokování a odstraňování objektů z paměti pomocí klíčových slov a příkazů daného jazyka, které vkládá do kódu programu již během

programování. Druhým způsobem je správa automatická, která využívá garbage collectory. Zde sice má programátor moc nad vytvářením objektů, ale jejich odstraňování je již zcela v rukou konkrétního garbage collectoru. Ten pomocí metod, které budou popsány dále sám vyhledá objekty, které již nejsou dostupné z kořenů a odstraní je.

Automatická správa paměti má tu výhodu, že programátorovi značně ulehčuje práci při psaní programu. Manuální správa totiž vyžaduje od programátora značnou kázeň a cit. Pokud odstraní objekt příliš brzo může se stát, že program nebude fungovat správně nebo dokonce nebude fungovat vůbec. Naopak použije-li příkaz pro odstranění příliš pozdě, může program do té doby zhavarovat v důsledku nedostatku paměti.

Oproti tomu použití garbage collectoru zpomaluje běh aplikace, zatěžuje procesor i paměť, prodlužuje dobu překladu a proto se u nižších jazyků, kde nám jde hlavně o rychlost nepoužívá.

Je také třeba si uvědomit, že oba způsoby mají i společné problémy jako je např. fragmentace, protože manuální správa paměti není bez freelistů dost dobře možná.

2.2. Metoda počítání referencí

Proto abychom zjistili, které objekty na haldě jsou dostupné z kořenů a které ne, není nutné graf objektů procházet pomocí některé z metod pro průchod grafu. Zcela nám postačí, když si u každého objektu na haldě budeme udržovat záznam, který nám poví kolik jiných objektů obsahuje ve svém těle odkaz na tento jeden konkrétní objekt. Vždy tedy, když se na náš objekt vytvoří nová reference, zvýšíme hodnotu uloženou v tomto záznamu o jedna a naopak, pokud bude některý s objektů, které na náš objekt odkazují odstraněn, sníží se hodnota záznamu rovněž o jedna. Je zřejmé, že pro uchování množství referencí na objekt poslouží skvěle jednoduchý čítač, který se tak stane nedílnou součástí každého objektu na haldě. V momentě, kdy jeho hodnota klesne na nulu, bude zřejmé, že k našemu objektu nevedou již žádné cesty a on bude odstraněn (přemístěn na freelist).

Tomuto způsobu garbage collectingu se říká *počítání referencí*. Na rozdíl od Mark-and-Sweep či Kopírovacího algoritmu, které budou popsány dále, tento algoritmus nepřerušuje chod programu svou činností, která je naopak rovnoměrně rozložena do celého běhu aplikace. Na druhou stranu, musíme u každého objektu udržovat čítač pro počítání referencí, což nás stojí paměť navíc a alokovat objekt je nyní podstatně složitější. Namísto prosté jedné strojové instrukce *x.fi* ← *p*, musí program vykonat:

```
z       ← x.fi
c       ← z.count
c       ← c - 1
z.count ← c
if c = 0 call putOnFreelist
x.fi    ← p
c       ← p.count
c       ← c + 1
p.count ← c
```

Ani snižování čítače není zcela bez problému. Při odstraňování objektu z haldy totiž může dojít ke spuštění celé laviny odstraňování. Pokud totiž při snížení čítače objektu dosáhneme nuly a objekt následně odstraníme může to vést k odstranění dalších objektů na haldě a to pak může vést ještě dál. V krajním případě se dokonce může stát, že budou odstraněny téměř všechny objekty. To by, ale znamenalo výraznou zátěž pro procesor a vedlo k značnému zpomalení běhu aplikace. Kvůli tomu se čítače u objektů, jejichž adresy byly uloženy v těle právě odstraněného objektu, snižují až s určitým zpožděním, aby se tak zabránilo řetězovému odstraňování.

Protože použití prostého čítače referencí může být extrémně drahé, je mnoho inkrementačních a dekrementačních operací eliminováno použitím analýzy datového toku. Například jakmile získáme hodnotu ukazatele skrz lokální proměnné, může překladač shrnout mnoho změn do jediného zvýšení či snížení hodnoty čítače nebo dokonce (pokud jsou výsledné změny nulové) nepoužít vůbec žádné instrukce. Nicméně i tak bude překladač vykonávat spoustu inkrementací a dekrementací čítače referencí, což jsou velmi drahé operace.

Metoda počítaní referencí by fungovala zcela ideálně za předpokladu, že by grafem objektů na haldě byl strom. Jenže graf objektů alokovaných na haldě stromem často nebývá a právě v cyklech mezi jednotlivými objekty dlí největší slabina této metody. Garbage collector, který využívá metody počítání referencí k odhalení mrtvých objektů, totiž nedokáže tyto cykly, či lépe řečeno objekty v těchto cyklech, odstranit. Jejich čítače totiž budou vždy alespoň na jedničce, takže budou pro collector stále živé, i když už to dávno nemusí být pravda a budou tak trvale zabírat místo v paměti.

Řešení problému s cykly mezi objekty

Jsou zde prakticky tři řešení jak v počítání referencí rozbít cykly. Ani jedno z nich však není ani jednoduché ani příliš elegantní.

1. Můžeme využít algoritmu Mark-and-Sweep či Stop-and-Copy, které jednou za čas projdou haldu a uvolní cykly. Zde se nabízí otázka proč pak správu paměti nenechat rovnou na jednom ze zmíněných algoritmů, navíc ztratí referenční čítač svou výhodu rozložení činnosti po celý běh programu.
2. Můžeme použít tzv. detektor cyklů, který projde celou hladu podobně jako Mark-and-Sweep, ale nebude odstraňovat objekty, pouze vyhledá cykly mezi nimi a pak jednu z vazeb zruší, čímž cyklus rozpojí a objekty budou následně odstraněny standardním způsobem. Hrozí zde ale možnost, že takto budou odstraněna i data, která jsou ještě dostupná.
3. Poslední řešení spočívá v rozlišování dvou druhů referencí – tzv. „silných“ a „slabých“ – a použití různých počitadel pro každý druh. Platí při tom, že pro zjištění, zda je objekt živý, jsou relevantní pouze silné reference. Toto řešení bohužel v některých případech nepracuje správně, protože i zde muže dojít k odstranění objektů, které by byly dostupné.

2.3. Mark-and-Sweep algoritmus

Jedinou variantou pro počítání referencí je průchod grafem objektů uložených na haldě. Pokud projdeme grafem tak, že za výchozí bod jsi zvolíme programové proměnné, navštívíme při tomto průchodu jen ty objekty, které jsou dostupné a jestliže jsi přitom tyto objekty nějakým způsobem označíme, můžeme pomocí dalšího průchodu, který bude tentokrát sekvenční, zkontrolovat všechny objety na haldě, a ty které nejsou označeny odstranit.

Algoritmu, který toto umí se říká *Mark-and-Sweep* algoritmus a je to, společně s počítáním referencí, nejstarší metoda garbage collectingu. Na rozdíl od počítání referencí nemá Mark-and-Sweep problémy s cykly a jedná se o tzv. stop the world algoritmus, což znamená, že je běh aplikace přerušen vždy, když je volán garbage collector. Obě metody využívají freelistů, a proto také trpí neduhem s tím spojeným a sice fragmentací, která je i hlavním problémem Mark-and-Sweepu, podobně jako velká zátěž, kterou pro běh aplikace tento algoritmus představuje, protože na rozdíl od jiných prochází haldu vždy dvakrát z toho alespoň jednou celou, což značně prodlužuje jeho trvání a brzdí tak běh aplikace.

Jak vyplívá z prvního odstavce dělí se tento algoritmus na dvě fáze. Ve fázi mark dojde k průchodu grafu živých objektů haldy, k tomu se nejčastěji používá algoritmus hledání do hloubky

(viz. Alg. 2.1), a k jejich o značení. Ve fázi sweep je pak halda procházena sekvenčním způsobem, kdy jsou jednotlivé objekty testovány zda jsou označeny, pokud ne dojde k jejich přesunutí na freelist.

```
function DFS(x)
    if x je ukazatel na haldu
        if záznam x není označen
            označ x
            for každé pole fi v záznamu x        // pole fi reprezentuje objekt v paměti,
                                                 // na který je ukazováno z x
                DFS(x.fi)
```

Alg. 2.1.: Prohledávání do hloubky

Použití zpětného ukazatele

Použití rekurze pro vyhledání a označení dostupných bloků paměti tak, jak nám to ukazuje algoritmus 2.1, je sice elegantní, ale může sebou přinést i problémy. S každým voláním totiž dochází k narůstání zásobníku volání, což může vést, v případě že je graf hlady rozsáhlý, až k jeho přetečení.

Nejednodušším řešením, které se nabízí je odstranění rekurze a převedení do iterativní formy. Použijeme zásobník, jiný než zásobník procesů, ve kterém uložíme objekty čekající na označení. Nejprve uložíme všechny objekty, které jsou v kořenech. Pak se dostaneme k samotnému označování objektů, nejprve označíme daný objekt a pak uložíme všechny jeho potomky na zásobník. Toto řešení nám ušetří volání funkcí, nicméně stále budeme spotřebovávat paměť alespoň tak velkou jako je výška grafu haldy.

Existuje však ještě lepší řešení jak dosažitelné objekty na haldě označit při využití konstantní paměti.

```
function DFS(x)
    if x je ukazatel a zároveň záznam x není označen
        t = null
        označ x; done[x] = 0
        while true
            i = done[x]
            if i < # polích v záznamu x
                y = x.fi
                 if y je ukazatel zároveň záznam y není označen
                        x.fi = t; t = x; x = y
                        označ x; done[x] = 0
                else
                        done[x] = i + 1
            else
                y = x; x = t
                if x == null then return
                i = done[x]
                t = x.fi; x.fi = y
                done[x] = i+1
```

Alg 2.2.: DFS využívající metody zpětného ukazatele

Algoritmus 2.2 nám ukazuje, že pro uložení objektů čekajících na označení se místo zásobníku dají využít samy objekty, takže pomyslný zásobník bude rozprostřen po grafu objektů, které čekají na dokončení. Po dokončení objektu zjistíme, existuje-li objekt o úroveň výš, a pokud ano, vrátíme se do něj a pokračujeme s jeho procházením. Tato verze „markování“ má konstantní paměťovou složitost, předchozí měla složitost lineární.

Negativem je, že ušetřené místo při procházení je vykoupeno prostorem, který musíme vyhradit pro skladování informace o aktuálním prvku každého objektu – už nestačí jediný bit, který by značil, jestli objekt byl nebo nebyl navštíven, ale je zapotřebí celé číslo, které nám udá kolik potomků objekt má.

Cena algoritmu Mark-and-Sweep

Časová náročnost algoritmu, ale hlavně SWEEP fáze, je dána množstvím dosažitelných dat a velikostí haldy, na které jsou uloženy. Předpokládejme, že máme *R* slov v dosažitelných datech na haldě o velikosti *H*. Pak je cena jednoho použití algoritmu Mark-and-Sweep dána vzorcem $c_1 R + c_2 H$, kde c_1 a c_2 jsou konstanty. Pakliže chceme zjistit úspěšnost průchodu, tedy jaké daný průchod uvolnil prostředky, vzhledem ke spotřebovanému času. Vydělíme spotřebovaný čas získaným smetím. Tedy:

$$\frac{c_1 R + c_2 H}{H - R}$$

Je-li *R* příliš blízké *H*, znamená to že se cena stala příliš velkou a při průchodu bude získáno jen malé množství dat. Pokud je *H* mnohem větší než *R*, pak je cena za alokaci slova přibližně rovna hodnotě c_2. Takto získaná hodnota může překladači říct zda-li je dobrý čas pro sběr mrtvých objektů.

2.4. Kopírovací algoritmus

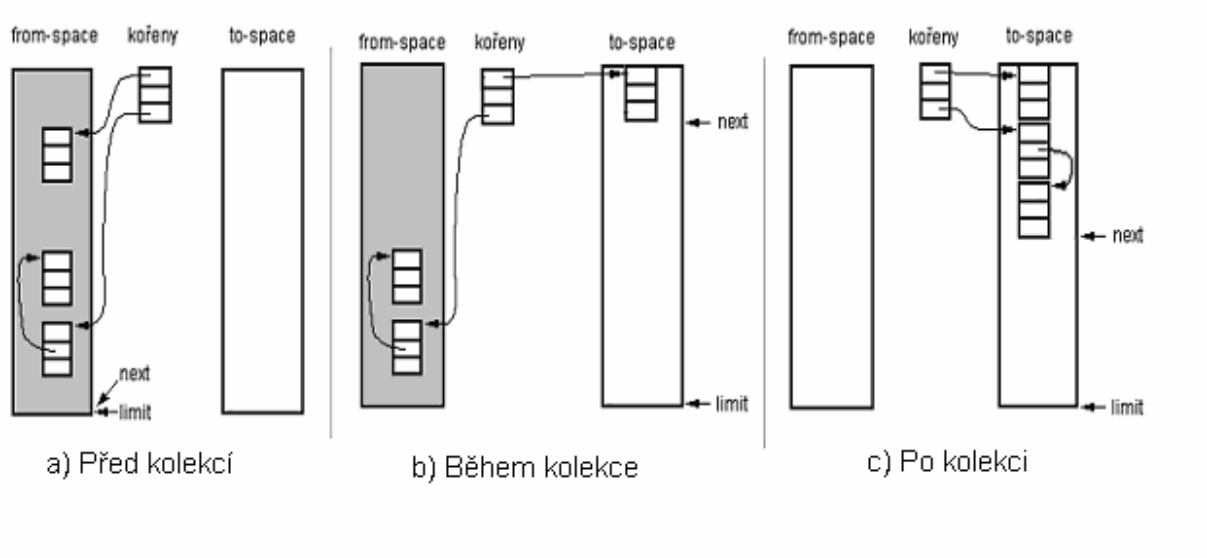
Kopírovací metoda, někdy nazývána také jako *Stop-and-Copy*, prochází grafem objektů, které jsou uložené na haldě, podobně jako metoda Mark-and-Sweep. Na rozdíl od ní tato metoda objekty na haldě neznačí, ale kopíruje je do předem určené oblasti v paměti. Při použití této metody se totiž paměť programu rozdělí na dvě stejně velké části, na tzv. from-space, kde se alokují nové bloky paměti, a to-space, který slouží právě jako cílová oblast při kopírování objektů během průchodu garbage collectoru. Díky tomu má ale kopírovací algoritmus k dispozici jen polovinu paměti a bude tak prováděn dvakrát tak častěji než ostatní metody garbage collectingu. Nicméně na rozdíl od Mark-and-Sweep mu stačí jediný průchod haldou na to, aby se zbavil mrtvých objektů, přičemž je nutno zmínit, že nevyužívá freelistů a netrpí proto ani fragmentací. Nicméně stejně jako u Mark-and-Sweep se jedná o stop the world algoritmus a dochází u něj tedy k přerušení běhu programu.

Na obr. 2.1 je znázorněna halda před a po použití kopírovacího algoritmu. Z obrázku je patrné, že metoda využívá dvou ukazatelů pro určení velikosti volného místa na haldě. Ukazatel limit míří na konec haldy, zatímco next ukazuje na začátek volného místa či lépe řečeno na konec posledního objektu v paměti.

Jak jsem zmínil výše, během průchodu garbage collectoru haldou jsou jednotlivé objekty kopírovány z from-space do to-space. Přenos probíhá tak, že se ve to-space alokuje blok paměti s identickou hlavičkou a o stejné velikosti jakou má originál a do jeho těla se zkopíruje obsah prvního pole originálu, zatímco do prvního pole v těle originálu se uloží adresa kopie. Poté se pokračuje dalšími bloky, dokud není zkopírována celá dosažitelná část haldy. Následně se zkopírují i ta pole, která nebyla přesunuta napoprvé. Místo původních adres bloků se do těl bloků zapíšou adresy nové,

které jsou uloženy v prvních polích originálů ve from-space, nakonec se zamění adresy i u ukazatelů z kořenů.

Alokace se zde provádí jednoduše tak, že se ukazatel next posune o požadovanou velikost směrem k limitu (jedná se tedy o onen nejjednodušší způsob alokace objektů na haldu). Dokonce i při průchodu garbage collectoru jsou jednotlivé kopie objektů alokovány stejným způsobem, jak je to vidět i na obr. 2.1.



obr. č. 2.1. Kopírovací algoritmus

Cena kopírovacího algoritmu

Podobně jako u Mark-and-Sweep, i zde se nám může hodit znát poměr mezi velikostí odstraněných (odstranitelných) objektů a velikostí haldy, abychom mohli určit účinnost sběru a tedy se i účině rozhodnout kdy je vhodné se sběrem začít.

Protože zde není žádná sweep fáze, a halda je rozdělena na dvě stejné části je amortizační cena rovna: $\frac{c_3 R}{\frac{H}{2} - R}$ instrukcí za jedno alokované slovo.

Pokud se H stane mnohem větším než R, přiblíží se cena nule. Realističtější je ale nastavení, při kterém H = 4R, kde je cena přibližně deset instrukcí na slovo. Což je, ale poněkud nákladné na čas i prostor. Pro snížení nákladů můžeme využít generační gabage collector.

2.5. Generační garbage collector

Na rozdíl od předchozích tří metod není generační garbage collector metodou, která by sama představovala jeden ze způsobů řešení garbage collectingu, ale jedná se rozšíření předchozích řešení.

Postupem doby se totiž zjistilo, že životnost objektů na haldě podléhá určitým pravidlům a sice, že mladší nedávno alokované objekty mají tendenci zanikat záhy po svém vzniku, zatímco objekty, které přežili již několik průchodů garbage collectoru, jsou dostupné zpravidla ještě dlouhou dobu. Jednoduše řečeno, bylo zjištěno, že pro zvýšení efektivity práce garbage collectoru by bylo

dobré zaměřit se na sběr mladých objektů. Z toho důvodu je nutné mladé a staré objekty od sebe rozlišovat, proto se zavedl systém generací.

Halda se, podobně jako u kopírovacího algoritmu, rozdělí, ale tentokrát ty díly nejenže nemusí být pouze dva, protože počet generací (částí haldy) může být libovolný, ale je dokonce nežádoucí, aby byly stejně velké. Z praxe totiž víme, že je lepší, když jsou starší generace větší, protože u nich neprobíhá garbage colleting tak často a naopak jsou do nich přehrávány objekty z nižších generací.

Protože jednotlivé generace mají své vlastní haldy, které vznikly rozdělením paměti pro celý program, je možné pro sběr různých generací využít různé metody garbage collectingu.

Sběr jednotlivých generací nicméně představuje problém již při své inicializaci. U generačního garbage collectoru je totiž množina kořenů rozsáhlejší než pouhé programové proměnné. Je nutné zajistit i ukazatele z ostatních generací, z pravidla se jedná o ty starší, protože je nepravděpodobné, že by mladší objekt ukazoval na starší. To ale není triviální operace. Když totiž budeme prohledávat starší generace, kvůli zajištění kořenů pro sběr mladší generace, zahodíme tím veškeré výhody získané použitím generací. Např. při každém sběru nejmladší generace bychom museli prohledat celou paměť a nejstarší generaci bychom prohledávali při sběru každé generace, takže místo toho, abychom počet průchodů snížili a ulevili tak zátěži, by počet průchodu naopak oproti bezgeneračnímu přístupu vzrostl.

Z toho důvodu se využívají tzv. Remmebered sety, ve kterých jsou uloženy adresy objektů, které odkazují do jiných generací.

Cena generačního sběru

V praxi je pro nejmladší generaci běžné, že obsahují méně než 10% živých dat. U kopírovacího algoritmu to znamená, že je H/R v této generaci rovno 10. Takže amortizační cena za získání jednoho slova je $\frac{c_3 R}{10R - R}$, tedy přibližně jedna instrukce. Sběr starších generací může být ale ještě dražší. Abychom se vyhnuli potřebě příliš velkého místa, použijeme pro starší generace menší poměr mezi H a R. To sice zvýší časovou náročnost sběru, nicméně ten je u starších generací dostatečně vzácný, takže celková spotřeba času zůstane dobrá.

Je také třeba si uvědomit, že údržba množiny ukazatelů ze starší generace do mladší rovněž spotřebovává čas. Takže pokud program mnohem více objektů aktualizuje než vytváří, může cena generačního garbage collectoru překročit spotřebu negeneračního.

2.6. Inkrementální algoritmus

Podobně jako generační i inkrementální garbage collector je rozšířením pro ostatní metody garbage collectingu. Tato metoda se snaží přenést jednu z největších výhod metody počítání referencí na stop the world algoritmy. Jinými slovy, využitím principů inkrementačního garbage collectoru rozložíme práci garbage collectoru více do běhu programu. Zjednodušeně řečeno to funguje tak, že se běh garbage collectoru přeruší, program vykoná potřebnou činnost a garbage collector pak pokračuje, kde skončil.

Abychom ale mohli navázat na činnost garbage collcetoru tam, kde jsme skončili je nutné, uchovávat změny v grafu dostupných dat. O to se stará tzv. *mutator*, jelikož změnám lze rozumět i jako mutacím grafu. Nejedná se o nic jiného než o zvláštní podprogram, který zároveň řídí běh garbage collectoru, protože ten u inkrementačního algoritmu pracuje pouze tehdy, pokud jej o to mutator požádá.

Aby garbage collector měl pehled o tom, které objekty již navštívil nebo které právě zpracovává případně, které byly během přerušení změněny používá se tzv. trojbarevný systém.

```
while pokud jsou zde nějaké šedé objekty
        vyber šedý záznam p
        for každé pole fi v záznamu p
                if záznam p.fi je bílý
                        obarvi záznam p.fi šedou
        obarvi záznam p černě.
```

Alg. 2.5.: Základní trojbarevné značení

Trojbarevné značení

V MARK & SWEEP a kopírovacím algoritmu se používají tři třídy záznamů:

1. **Bílé** = objekt ještě nebyl navštíven garbage collectorem. Bílé objekty na konci cyklu jsou smetím a budou odstraněny.
2. **Šedé** = objekt již byl navštíven (označen či zkopírován)**,** ale jeho potomci nebyly ještě označeni. Případně byl během minulého přerušení pozměněn a musí být znovu zkontrolován. V MARK & SWEEP se jedná o objekt v zásobníku
3. **Černé** = objekt byl označen i s potomky a garbage collector se k němu již nebude během tohoto cyklu vracet. V MARK & SWEEP byl již tento objekt vyzvednut ze zásobníku.

Všechny objekty jsou na začátku pochopitelně bílé. Pomocí algoritmu 2.5. collector začerní šedé objekty zatímco jejich bílé potomky označí šedou barvou. Změně barvy z šedé na černou přitom dopovídá vyzdvihnutí objektu ze zásobníku a podobně změna z bílé na šedou zase uložení na zásobník. Celý algoritmus končí ve chvíli, kdy jsou všechny objekty v paměti buď černé nebo bílé, přičemž bílé jsou smetí a budou odstraněny. Z výše uvedeného plyne, že:

1. Žádné černé objekty neodkazují na bílé
2. Všechny šedé jsou součástí datové struktury collectoru(jsou na zásobníku)

Pokud mutator, při vytváření objektu během sběru a následné aktualizaci pole ukazatelů existujících objektů, poruší jedno z těchto pravidel, nebude sběrný algoritmus fungovat.

2.7. Boehmův algoritmus

Celým názvem Boehm-Demers-Weiserův algoritmus je zástupcem tzv. konzervativních collectorů. To znamená, že tento algoritmus nepotřebuje spolupráci s kompilátorem a je ke kódu překladače přiložen jen jako knihovna, která poskytuje své funkce pomocí rozhraních metod.

Boehmův algoritmus je postaven na algoritmu Mark-and-Sweep, ale poskytuje přitom i výhody inkrementačního a generačního algoritmu. Na rozdíl od klasického Mark-and-Sweep algoritmu, ale Boehmův algoritmus umožňuje při odstraňování objektů z paměti vyvolat finalizační kód, který umožňuje provést, ještě těsně před odstraněním objektu, nějaké operace.

Algoritmus pracuje ve čtyřech fázích:

1. **Přípravná fáze** – vynuluje všechny mark bity, aby bylo zřejmé, že všechny objekty jsou na začátku potencionálně nedosažitelné.
2. **Mark fáze** – za pomoci procházení grafu haldy skrze řetězce ukazatelů označí všechny dosažitelné objekty.
3. **Sweep fáze** – vyhledá na haldě všechny nedosažitelné a tudíž neoznačené objekty a vrátí je na příslušný free list pro znovu použití.
4. **Finalizační fáze** – nedosažitelné objekty, které se zaregistrovaly pro finalization, jsou seřazeny vně collectoru. Následně jsou vykonány příslušné finalizační operace.

Boehmův algoritmus je napsán tak, aby jeho metody umožňovaly také správu paměti pomocí vláken či paralelního zpracování. Jeho velkou výhodou je jeho všestrannost.

Algoritmus lze téměř bez úprav použít na libovolném operačním systému a pro implementaci libovolného programovacího jazyka. Univerzálnost tohoto algoritmu je nicméně vykoupena jeho vysokou složitostí, některé metody se při určitých nastaveních jako jsou např. podpora vláken či paralelní zpracování chovají různě, také používaný operační systém může vyžadovat pro správný běh metody její odlišnou implementaci, kód tohoto algoritmu s tím vším musí počítá je proto mnohem komplikovanější než běžná implementace Mark-and-Sweepu.

Popisem Boehmova algoritmu končí druhá kapitola, ve které jsme si popsali obecné principy fungování algoritmů, které se používají pro garbage collecting. V následujících kapitolách vám přiblížím konkrétní implementace daných algoritmů v implementacích tří různých programovacích jazyků.

3. OCaml

OCaml, což je zkratka pro Objective Caml, je jednou ze dvou implementací jazyka Caml, který byl vyvinut roku 1985 ve Francii. OCaml je staticky typovaný funkcionální jazyk, který byl rozšířen o možnosti objektově orientovaného programování. Zde popsaná implementace tohoto jazyka je verze 3.10.2. OCaml má jak interpret tak překladač a jeho garbage collector využívá metod Mark-and-Sweep a kopírovacího algoritmu ke sběru nedostupných objektů.

3.1. Popis struktury paměti jazyka Ocaml

Paměť je v jazyce OCamlu rozdělena na dvě části, na minoritní a majoritní haldu. Toto rozdělení paměti na dvě různě velké části (minoritní halda je podstatně menší než majoritní) dává tušit, že garbage collector, který je zde použit bude využívat vlastností generačního garbage collectoru. Nyní si ale popišme vlastnosti obou částí paměti.

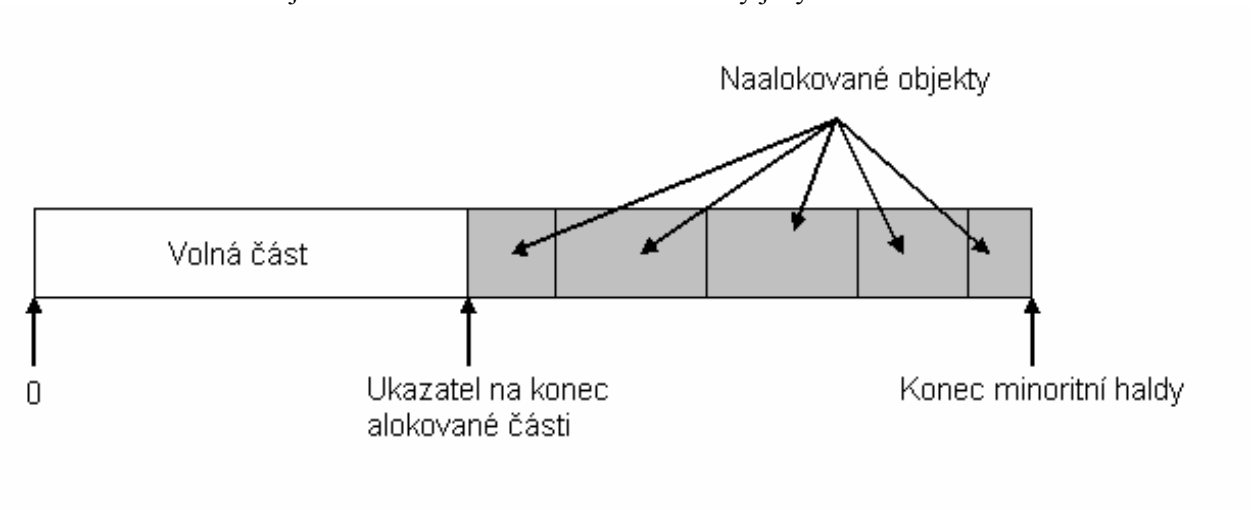
Minoritní halda

Minoritní halda je první zastávkou v životním cyklu objektu jazyka OCaml, které jsou dostatečně malé, aby na ni mohly bát alokovány.

Minoritní halda je vytvořena na začátku běhu programu a její délka je po celou dobu běhu neměnná a nastaví se pomocí metody *caml_set_minor_heap_size* ve třídě minor_gc.c. Minoritní halda OCamlu má, ostatně jako každá halda, která je spravována pomocí algoritmu Stop-and-Copy, dvojici ukazatelů a je zaplňována od konce (od nejvyššího bitu k 0). První z ukazatelů odkazuje na konec zaplněné části, zatímco druhý odkazuje na nejvyšší bit tedy na začátek haldy.

Alokace probíhá tak, že se ukazatel, který ukazuje na konec posledního prvku, a který je v implantaci jazyka pojmenován jako *caml_young_ptr*, posune o velikost alokovaného prvku blíže k začátku haldy, protože velikost alokovaného prvku odečte od jeho aktuální pozice. Jestliže by pak adresa, na kterou by *caml_young_ptr* po alokování objektu ukazoval, byla záporná, je jasné že nově alokovaný objekt se na haldu nevejde a je nutné, za pomoci garbage collectoru, provést vyčištění haldy.

Na obrázku 3.1. je znázorněna struktura minoritní haldy jazyka Ocaml.



obr 3.1.: Struktura minoritní haldy

Majoritní halda

Majoritní halda jazyka Objective Caml je druhou etapou v životním cyklu objektu tohoto jazyka. Jde vlastně o spojený seznam tzv. *chunků*, ve kterých jsou pak alokovány samotné objekty Objective Camlu. Rozměr těchto chunků je pevně dán, ale je-li vyčerpán lze alokovat další nové chunky a to a ž do plné paměti. Rozměr chunků je roven násobku velikosti stránky paměti.

Chunky se skládají z hlavičky a těla, v němž jsou objekty alokovány. Ukazatelé pak odkazují na nultý prvek v těle chunku místo jeho hlavičky.

Majoritní halda neslouží jen jako to-space pro minoritní haldu, ale alokují se na ní i nové objekty, které jsou příliš velké než aby mohli být alokovány na minoritní haldu.

Na rozdíl od minoritní hlady využívá majoritní halda při alokování objektů freelist volných objektů. Freelist zde tvoří seznam volných nebo uvolněných objektů. Ukazatel freelistu zde ale neukazuje na jeho začátek, nýbrž na poslední záznam freelistu, jenž byl na posledy použit (či lépe řečeno na ukazuje na ten další). Freelist je udržován setříděný, což je důvod proč si freelist pamatuje, který blok (či lépe řečeno jak velký blok) paměti naposledy poskytl, je totiž velká pravděpodobnost že další alokace bude potřebovat blok podobné velikosti, takže vyhledat blok odpovídající požadavkům pak není tak náročné. Pokud, ale freelist blok požadovaných parametrů neobsahuje, je buď alokován nový chunk nebo zavolán garbage collector.

Při alokaci nového bloku mohou nastat tři případy:

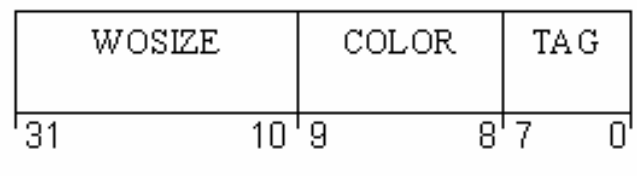
1. Nalezený volný blok je přesně tak velký, jak bylo požadováno. Blok je odpojen z freelistu a vrácen.
2. Nalezený blok je o jedno slovo delší než je požadovaná velikost. Blok je uvolněn z freelistu. Zbývající slovo, ale nemůže být připojeno zpět, takže se změní na prázdný blok (bude obsahovat jen hlavičku viz. Popis objektů v Objective Camlu) a vrátí se zbytek.
3. Nalezený volný blok je velký víc než dost. V tomto případě dojde k rozdělení bloku na dva a navrácení pravého bloku, který má požadované rozměry.

Samotný freelist nestojí, jak by se mohla zdát, mimo obě haldy, ale je nedílnou součástí majoritní haldy v jejichž chuncích se ukrývá. Pro rozlišená bloků haldy od bloků freelistu při sweep fázi mají bloky freelistu modrou barvu.

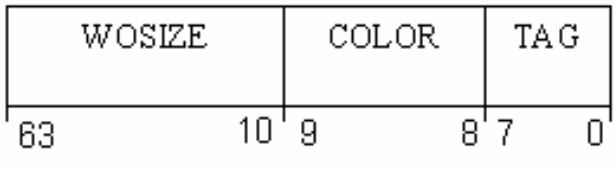
Popis objektů v Objective Camlu

Každý objekt v OCamlu je tvořen hlavičkou a tělem v němž jsou uložena data. O struktuře a způsobu uložení těchto dat v těle objetu, rozhoduje to jakého typu daný objekt je, což je vlastnost určená hodnotou pole *tag* v hlavičce objektu.

Na obr. č. 3.2. a 3.3. je vidět struktura hlavičky pro 32 bitovou respektive 64 bitovou architekturu.



obr. 3.2. Struktura hlavičky v 32b arch



obr. 3.3. Struktura hlavičky v 64b arch

První část udává velikost bloku ve slovech. Druhá část informuje garbage collector o tom v jakém stádiu sběru se blok nachází (garbage collector OCamlu rozeznává čtyři barevně rozlišené stavy, ve kterých se může blok paměti nacházet). Pole tag nám pak říká jaký druh dat je v bloku obsažen.

Ocaml rozlišuje deset speciálních tagů. Nás ale budou zajímat je některé z nich. Jedná se o tyto tagy:

1. Infix_tag 249
2. Forward_tag 250 – forwarding ukazatel, on a infix_tag se pro potřeby sběru minoritní haldy nacházejí hned u sebe
3. Abstract_tag 251 – Jedná se o nejnižší tag pro bloky které neobsahují žádnou hodnotu. Někdy taky No_scan_tag (přesněji oba tagy mají stejnou hodnotu).

3.2. Popis funkce garbage collectoru v jazyce OCaml

Jak již bylo řečeno výše, využívá OCaml generační garbage collector. To ale není a ani nemůže být jediná metoda, které OCaml vyžívá. Při pozorném pohledu na Minoritní haldu je jasné, že se zde využívá kopírovací algoritmus. Při pohledu na implementaci garbage collectoru, zde narazíme také na MARK & SWEEP algoritmus, který je využíván pro sběr odpadu v majoritní haldě, a který je vylepšen podporou Inkrementačního algoritmu. Ale začněme od začátku.

Popis průběhu minoritní kolekce

Když se při alokaci nového objektu zjistí, že na minoritní haldě již není dostatek místa, je volána metoda *caml_minor_collection*, která zahájí činnost garbage collectoru pro sběr minoritní haldy. V těle této metody je pak volána, a to hned dvakrát, jednou na začátku a jednou na konci, to abychom si mohli být jistí, že je minoritní halda skutečně prázdná, metoda *caml_empty_minor_heap*, která se skrze volání metod *caml_oldify_local_roots* a *caml_oldify_one* postará o samotné přesunutí dosažitelných objektů do majoritní haldy. Na začátku metody *caml_empty_minor_heap* se kvůli jejímu druhému volání na konci sběru zjišťuje, zda je minoritní halda prázdná či nikoliv, aby nedocházelo ke zbytečnému volání metod. Samotný proces přesunu bloků je naimplantován v metodě *caml_oldify_one*, která je volána i z metody *caml_oldify_local_roots*.

Garbage collector nejprve projde objekty dosažitelné z kořenů a pak teprve ty objekty, které jsou dosažitelné z majoritní haldy. Zkopírování objektu probíhá tak, že je na majoritní haldě alokován objekt stejné velikosti a se stejnou hlavičkou, jako objekt který garbage collector právě zpracovává v minoritní haldě. Nedojde ale ke zkopírování všech polí objektu, výjimku tvoří objekty s tagem No_scan_tag, ale je přenesena pouze hodnota nultého pole z těla objektu. Hlavička originálu objektu v minoritní haldě je vynulována a do nultého pole originálu je uložena adresa kopie, o takto ne zcela přesunutém objektu je pak vytvořen záznam v *oldify_todo_list* a jeho přesun bude dokončen voláním metody *caml_oldify_mopup*, která je volána z těla metody *caml_empty_minor_heap*. Pokud měl blok jen jedno pole přesune se pochopitelně celý najednou.

Během přesunu může dojít ke třem situacím:

1. Hodnota pole tag v hlavičce objektu je menší než Infix_tag pak vše probíhá normálně
2. Je-li tag roven Infix_tag pak bude na místo daného bloku přesunut blok následující, tedy takový blok který byl alokován hned po bloku s Infix_tagem a je tedy v paměti hned za ním.
3. Hodnora tagu je No_scan_tag nebo vyšší. Pak je blok přesunut celý, nehledě na jeho velikost a počet polí v těle.
4. Pokud je tag roven hodnotě Forward_tag, pak se bude nejprve zkoumat nulté pole dané bloku B1. Pokud obsahuje ukazatel na blok paměti (říkejme mu blok B2) a pokud tag tohoto bloku je také Forward_tag či alespoň Lazy_tag nebo Double_tag pak se blok B1 přesune normálním způsobem. Pokud hodnota tagu v hlavičce bloku B2 není rovna jednomu ze zmíněných tagů nebude se blok B1 přesouvat, ale místo toho se běžným způsobem zpracuje blok B2.

Jakmile jsou objekty přesunuty, je z těla metody *caml_empty_minor_heap* volána metoda *caml_oldify_mopup*, aby přesun dokončila pro ty bloky, které měli ve svém těle více než jedno pole a byly uloženy na oldify_todo_list. V tomto seznamu jsou uloženy adresy originálních bloků v minoritní haldě. Seznam je sekvenčně procházen a pole daných bloků přesouvána do majoritní haldy (adresy kopii jsou uloženy v nultých polích daných originálních bloků). Pokud se při přesunu zjistí, že dané pole obsahuje ukazatel na blok paměti je daný blok zavolána metoda *caml_oldify_one* a blok je přesunut do majoritní haldy a původní ukazatel je přepsán, aby odkazoval na novou pozici objektu.

Na konci se caml_young_ptr nastaví na konec haldy, čímž se minoritní halda smaže. Ukazatel caml_young_limit se pak nastaví na začátek haldy, aby mohl, v případě nutnosti, signalizovat její přeplnění. V dalším kroku se smažou záznamy v tabulce, ve které byli uloženy adresy bloků dostupných z majoritní haldy. Proměnná *caml_in_minor_collection* se nastaví na 0 a dá nám tím signál, že vlastní minoritní sběr skončil

.

Ve metodě *caml_minor_collection* není spuštěn jen minoritní sběr, ale rovněž je zde, po vykonání minoritního sběru, volána metoda *caml_major_collection_slice* s parametrem 0, který značí, že garbage collector vypočítá množství práce, kterou na majoritním sběru vykoná a poté jej spustí.

Popis majoritní kolekce

Zatím co minoritní kolekce proběhne vždy celá na jednou, majoritní kolekce, která využívá inkrementačního přístupu a implementuje algoritmus Mark-and-Sweep, může být přerušována činností programu. Z toho důvodu rozeznává OCaml čtyři stavy objektů alokovaných v paměti. Jednotlivé stavy jsou rozeznávány pomocí proměnné *color* v hlavičce objektu. Ta může nabývat hodnot:

1. **Bílá**: objekt ještě nebyl garbage collectorem nalezen. Na konci cyklu jsou všechny bílé objekty smetím a budou odstraněny.
2. **Šedá**: objekt již byl collctorem nalezen ale ještě nebyl zcela zpracován (nebyli zpracováni jeho potomci) a/nebo byl collector přerušen během programu. Collector se k objektu ještě vrátí.
3. **Černá**: objekt byl zpracován. Collector se jím již nebude zabývat.
4. **Modrá**: objekt je součástí freelistu.

Spouštěcí metodou je *caml_major_collection_slice*, která v jednotlivých fázích cyklu garbage collectoru určí kolik práce je třeba udělat a s tímto parametrem pak volá jednotlivé funkce, která pak představují implementaci fází algoritmu Mark-and-Sweep. Na samém počátku je volána metoda

start_cycle, která nám spustí fázi mark a zavolá metodu *caml_darken_all_roots*. Ta postupně označí všechny kořeny majoritní haldy.

Fáze kolekce je nastavena na mark, pod-fáze je main a z těla metody *caml_major_collection_slice* je volána metoda *mark_slice*, které se jako parametr předá hodnota kolik práce se má vykonat. Množství práce se rovná velikosti zpracovaných bloků ve slovech.

V pod-fázi main dochází k samotnému označování dosažitelných objektů. Celá fáze mark je vložena do cyklu while, jehož podmínkou je kladná hodnota proměnné *work*, která přestavuje množství požadované práce vypočtené v předchozí metodě. Během vykonávání metody se postupně procházejí jednotlivé chunky a v nich obsažené objekty. Ty jsou během prvního průchodu obarveni šedou barvou, teprve až jsou zpracováváni jejich potomci, jsou obarveni černě. Během zpracování objektu ještě před jeho obarvením je podobně jako v minoritní haldě testována jejich hlavička na tag. Testují se vždy potomci objektů, které byly zpracovány minule. Pokud se jedná o první průchod jsou tetovány objekty, které jsou přímo dostupné z kořenů. Zde jsou dvě možnosti:

1. Tag je Forward_tag. V takovém případě je otestován i potomek tohoto potomka (říkejme mu B) tohoto objektu na podmínku shodnou s podobným případem u minoritní haldy. Tedy pokud je tag tohoto potomka B roven Forward_tag nebo Lazy_tag nebo Double_tag bude zpracován i původní potomek. Jinak se potomek přeskočí a na jeho místo se zapíše adresa jeho potomka B, který se pak bude dal zpracovávat.
2. Tag je Infix_tag. Pak se bude zpracovávat blok paměti sousedící se zkoumaným potomkem.

Poté se objekty obarví šedou barvou, aby na začátku dalšího průchodu mohl být obarven černě. Ukazatel *gray_vals_ptr*, který nám předtím určil objekt jehož potomky jsme zkoumali, se nastaví aby ukazoval na právě obarvený objekt.

Pod-fáze main skončila, objekty máme označené a můžeme přistoupit k pod-fázi weak1. Zde dojde k odstranění slabých ukazatelů na mrtvé hodnoty. Dochází zde k procházení tabulky slabých referencí. Ty jsou zpracovávány podobně jako předtím reference silné, tedy opět dochází k testování tagů s tím, že nás zde zajímá pouze Forward_tag. Pokud se při práci zjistí, že daná slabá reference vede k bílému objektu, je taková reference smazána a objekt je ponechán bílý.

Následuje pod-fáze weak2, ve které jsou odstraňována slabá mrtvá pole. Nyní se testuje, zda objekty obsažené v tabulce, tedy ty u kterých jsme v minulém kroku zjišťovali kam vedou jejich slabé reference, jsou bílé či nikoliv. Pokud ano je daný záznam z tabulky vymazán.

Nyní můžeme přijít k pod-fázi final. Kde se fáze collectoru nastaví na fázi sweep.

Metoda *swee_slice* je také volána z metody *caml_major_collection_slice* a má rovněž učeno množství práce, kterou musí vykonat. Garbage collector zde sekvenčně prochází celou haldu a testuje hlavičky nalezených bloků na jejich barvu. Černé bloky jsou obarveny na bílo. Bílé jsou pomocí volání metody *caml_fl_merge_block* připojeny seřazeným k blokům freelistu a obarveny na modro. Modré jsou garbage collectorem ignorovány. Na konci metody se potom ještě nastaví fáze collectoru na idle a proměnná work nastaví na nulu, aby se ukončil cyklus v němž je vše naimplementováno.

Ačkoliv bylo místo na hladě uvolněno a obě hlavní metody ukončeny, zbývá ještě jedna věc která musí být provedena. Jsme zpátky v těle funkce *caml_major_collection_slice* a kromě statistik, které nám metoda o právě ukončeném cyklu může nabídnout, je zde nutnost zkontrolovat fragmentaci haldy a zvážit její případnou defragmentaci pomocí metody *caml_compact_heap* (samotný výpočet nutnosti volání této metody je uveden v metodě *caml_compact_heap_maybe*, která je volána po skončení cyklu garbage collectoru).

Defragmentace haldy

Defragmentace majoritní haldy stojí zcela mimo tělo samotného collectoru a je prováděna teprve až po provedení kompletního cyklu garbage collectingu a to jen v případě, že fragmentace majoritní haldy dosáhla svých mezí. Je tedy vykonávána ještě méně častěji než samotná majoritní kolekce, která je zase volána méně častěji než kolekce minoritní.

Za samotnou defragmentací stojí metoda *caml_compact_heap*, která ve čtyřech průchodech přes celou majoritní haldu provede její defragmentaci.

1. Během prvního průchodu dojde k přezkoumání hlaviček všech objektů na haldě, tedy i těch, které jsou umístěny na freelistu, přičemž právě hlavičkám uvolněných objektů je během tohoto průchodu změněn tag na String_tag. Dále dojde k zakódování neinfixových hlaviček.
2. Druhý průchod haldou slouží k invertování ukazatelů a hlavně k spojení všech bloků do invertovaného seznamu s tím, že bloky s Infix_tagem jsou spojeny do vlastního seznamu a oddělení od ostatních.
3. Během třetího průchodu dojde k navrácení invertovaných ukazatelů, k dekódování hlaviček. Právě zde jsou z defragmentace vyloučeny ty chunky, ve kterých je příliš málo volného místa (děje se tak prostřednictvím metody *compact_allocate*).
4. Konečně během čtvrtého průchodu haldou dojde k realokaci a přesunu objektů, přičemž je zde znovu použita metoda *compact_allocate*, která nám zajistí, že budou zpracovány pouze ty chunky, které to potřebují.

Pokud se během defragmentace haldy zcela vyprázdnily některé chunky, pak budou tyto chunky uvolněny z paměti ihned po skončení čtvrtého průchodu. Z neprázdných chunků jsou sesbírána data o velikosti volného a využitého místa. Následně jsou procházeny prázdné chunky, které jsou uvolňovány, pokud je celková velikost volného místa na haldě větší než procentuální vyjádření místa využitého. Pokud je více využitého míst, je prázdný chunk ponechán s tím, že se jeho velikost přičte k celkové velikosti volného místa na majoritní haldě.

V posledním kroku celé metody je pak znovu vybudování freelistu, který byl kvůli defragmentaci smazán. Halda je zde znovu celá projita, přičemž volné místo v jednotlivých chuncích je rozsekáno na bloky a umisťováno na freelist.

Tím je popis grabage collectingu v implementaci jazyka OCaml skončen a tak může přejít k popisu implementace jazyka Python.

4. Python

Python je interpretovaným jazykem, což znamená, že jeho kód není překládán celý najednou, ale jednotlivé funkce napsané v kódu Pythonu jsou překládány až v momentě jejich zavolání. Tento způsob překladu je sice pomalejší, ale nabízí jednodušší ladění programu a odstraňování chyb. Podobně jako OCaml je i Python objektově orientovaný jazyk, ale s dynamicky typovanými daty. Na rozdíl od OCaml má Python celou řadu implementací. Zde je popsaná základní implementace Pythonu ve verzi 3.0. (někdy nazývána taky jako CPython)

4.1. Stručný popis objektů

Každý objekt má svůj typ, který reprezentuje jaký druh dat obsahuje. Typ je objektu pevně dán při vytvoření a typy samotné jsou reprezentovány jako objekty typu type. Objekty obsahují ukazatele na objekty reprezentující jejich typy, ty pak ukazují na objekt, který reprezentuje typ type a ten ukazuje sám na sebe. Objekty, které reprezentují typ objektů, a to včetně typu type se nedají z paměti odstranit, protože nemají čítač referencí. Je to pochopitelné, neboť bychom jinak buď nemohli objekty některých typů alokovat, protože objekt reprezentující jejich typ by v paměti již neexistoval a nebo bychom museli alokovat objekty dva.

Pro nás nejdůležitější součástí každého objektu je jeho čítač referencí, který se zvýší či sníží, když je ukazatel na objekt zkopírován nebo smazán. Jakmile klesne na nulu je objekt odstraněn z haldy. Další velmi důležitou součástí objektů je ukazatel *tp_clear***,** který je typu *inquiry*, a který odkazuje na čistící funkci, jenž ovšem neodstraní objekt, jak bychom se mohli domnívat, o to se totiž stará automaticky dekrementační makro, ale odstraní referenci na objekt a rozbije tak cyklus, ve kterém se objekt nachází. Tato funkce je dostupná, pouze u objektů, které nejsou typu immutable, u kterých se předpokládá, že cyklus nevytvoří, zde je *tp_clear* typu *NULL* a neodkazuje nikam.

Velikost alokovaného objektu je neměnná, pokud by obsahoval data s proměnnou délkou, bude takový objekt obsahovat ukazatele na různě velké části objektu, které budou obsahovat příslušné množství dat. Díky tomu může být reference na daný objekt reprezentována prostým ukazatelem.

Důležitým atributem každého objektu při provádění garbage collectingu je *gc_refs*, které během činosti detektoru cyklů může nabývat následujících hodnot:

1. **GC_UNTRACKED:** Počáteční stav. Objekty vrácené PyObject_GC_Malloc se nacházejí v tomto stavu.
2. **GC_REACHABLE:** Objekt je v nějakém generačním seznamu. Mohou být vysledováni jeho potomci. Objekt přechází do GC_REACHABLE pokud jej bylo dosaženo z vnějšku generace či z jiného GC_REACHABLE objektu
3. **GC_TENTATIVELY_UNREACHABLE:** Jen objekty se stále nastaveným GC_TENTATIVELY_UNREACHABLE jsou kandidáty pro sběr. Pokud je rozhodnuto nesebrat takový objekt, pak je jeho gc_refs znovu nastaveno na GC_REACHABLE.
4. **>= 0 :** Během průchodu collectoru přes generaci, jsou do gc_refs zkopírovány hodnoty čítačů referencí jednotlivých objektů. Od nich je pak odečteno číslo, které reprezentuje kolik referencí na daný objekt směřuje z vnějšku generace. Objekty, které poté mají gc_refs > 0 jsou přímo dostupné z vnějšku, zatímco objekty s gc_refs = 0 ne.

4.2. Popis činnosti garbage collectoru

Tato implementace jazyka Python se spoléhá na počítání referencí především proto, že Python je interpretovaným jazykem a počítaní referencí díky své schopnosti rozložit zátěž spojenou s používáním grabage collectoru téměř rovnoměrně do celého běhu programu, vyhovuje lépe než stop the world algoritmy, které by běh zatěžovali nárazově. Nicméně samotné počítání referencí nestačí. Jak bylo řečeno v druhé kapitole, má počítání referencí vážný problém s objekty mezi kterými se vytvořil cyklus, a které nedokáže odstranit. Tato implementace Pythonu se proto spoléhá na řešení pomocí detektoru cyklů, který využívá metodu podobnou algoritmu Mark-and-Sweep, přičemž zde proběhne jen fáze mark s tím, že se s její pomocí cykly detekují a poté pomocí volání *tp_clear* rozbijí a nakonec odstraní standardním způsobem pomocí volání dekrementačního makra, které pak zavolá dealokátor. Celý detektor cyklů je i s pomocnými metodami implementován ve souboru gcmodule.c, přičemž srdcem celého detektoru je metoda collect.

Podobně jako u OCamlu i zde je garbage collector rozšířen o generační přístup ke sběru mrtvých objektů. O ten se ve všech generacích stará metoda počítání referencí, která ale nemá vliv na pohyb objektů mezi generacemi. O přesun objektů z mladší generace do starší se stará výhradně detektor cyklů a pokud jej tedy vypneme, což je možné díky metodě disable(), pak o generace přijdeme.

Počítání referencí

Na rozdíl od OCamlu, v Pythonu není cyklus garbage collectoru spuštěn kvůli nedostatku místa na haldě, který byl zjištěného během pokusu o alokaci nového objektu. Zde totiž, diky využití metody počítání referencí, probíhá proces odstraňování mrtvých objektů neustále po celou dobu běhu aplikace. O volání dealokátoru se přitom stará makro Py_DECREF(op), které zajišťuje snižování čítače referencí s tím, že pokud jeho počet klesne na nulu zavolá objektový dealokátor a umístí daný objekt na freelist. Opakem je mu makro Py_INCREF(op).které naopak hodnotu čítače zvýší vždy, objeví-li se nová reference na daný objekt.

Při odstraňování objektů z paměti hrozí nebezpečí, že dojde ke spuštění lavinového efektu a nárazovému odstranění velkého množství objektů, což by silně zatížilo interpret jazyka a zpomalilo běh aplikace. Abych tomuto stavu předešli, je dobré rozbít takovýto řetězec na menší kousky, které tolik nezatíží běh programu.

```
define PyTrash_UNWIND_LEVEL 50

makro Py_TRASHCAN_SAFE_BEGIN(op)
        if (_PyTrash_delete_nesting < PyTrash_UNWIND_LEVEL)
                ++_PyTrash_delete_nesting;
                /* Zde je tělo dealokátoru. */
makro Py_TRASHCAN_SAFE_END(op)
                --_PyTrash_delete_nesting;
                if (_PyTrash_delete_later && _PyTrash_delete_nesting <= 0)
                        _PyTrash_destroy_chain();
                else
                _PyTrash_deposit_object((PyObject*)op);
```

Alg.: 4.1. Makra Trashcan mechanismu

V této implementaci jazyka Python se k tomu využívá tzv. Trashcan mechanismus. Ten "obalí" tělo dealokátoru makry, které se postarají o to, aby o odstranění nežádalo příliš mnoho objektů najednou. Ty makra se jmenují Py_TRASHCAN_SAFE_BEGIN(op) a Py_TRASHCAN_SAFE_END(op) a jsou znázorněny v algoritmu 4.1.

Makro BEGIN navýší čítač objektů, které dealokátor žádají o uvolnění z paměti. Dokud je hodnota čítače nízká funguje tělo dealokátoru normálně, to znamená, že objekty jsou z paměti uvolněny, jakmile o to požádají. Když se ale hodnota čítače zvýší a přeroste tak hodnotu *PyTrash_UNWIND_LEVEL*, pak se tělo dealokátoru přeskočí a objekt uloží na seznam objektů, které čekají na odstranění.

Po volání dealokátoru se požije makro END, které sníží počet objektů, které žádají o uvolnění, ale to se v žádném případě netýká objektů na seznamu čekajících, zde je pouze sníženo číslo, které značí počet objektů, které dealokátor zpracovává, teprve pak je volána funkce, která objekty uložené na seznamu znovu nabídne dealokátoru, pokud ten stále nebude moci počká se na uvolnění dalšího ze zpracovávaných objektů. Ve výsledku je řetězec N dealokací rozbit na $\frac{N}{\text{PyTrash\_UNWIND\_LEVEL}}$ kusů. Čímž se zabrání možným chybám zásobníku.

Detekce a odstraňování cyklů mezi objektů

Cykly mezi objekty jsou Achillovou patou počítání referencí. Zde je tento problém řešen pomocí detektoru cyklů, který je implementován v gcmodule.c. Na rozdíl od samotného počítání referencí je detektor volán ve stejnou chvílí, jako celý garbage collector u OCamlu, tedy v okamžiku. kdy alokátor při pokusu umístit objekt na haldu zjistí, že zde pro něj není dost místa.

Než začneme s popisem bylo by dobré ujasnit si několik termínů. Pokud budu v textu mluvit o množině objektů *young*, mám tím na mysli objekty v právě zpracovávané generaci, pojmem *old* myslím generaci o jedna starší, *unreachable* je seznam nedosažitelných objektů a *finalizers* je seznam objektů, které jsou přímo či nepřímo dosažitelné z objektů, a kterým se říká finalizers, jenž obsahují metody typu __del__.

Na začátku sběru dojde ke spojení young se všemi mladšími generacemi, což mimo jiné znamená, že pokud je young nejstarší generace, pak bude zpracována celá halda.

V dalším kroku, jsi určíme, které objekty v young jsou přímo dostupné z vně young, to znamená, že na ně odkazuje reference uložená v nějakém objektu, který se nenachází v young. Takové objekty v young totiž budou mít jako jediné *gc_refs* větší než nula, zatím co ostatní objekty, které se nacházejí v young budou mít *gc_refs* nulový. Toho je dosaženo voláním metod *update_refs* a *substract_refs*, kdy první z nich nastaví *gc_refs*, aby odpovídal hodnotě, která je uložena v čítači referencí, zatímco druhá metoda pak od *gc_refs* odečte počet referencí, které směřují na objekt z vnitřku young.

Následně dojde k oddělení nedosažitelných objektů od zbytku objektů v young. Tyto objekty se uloží na seznam unreachable a jejich *gc_refs* se nastaví na GC_TENTETIVELY_UNREACHABLE. V young tak zůstanou pouze objekty, která jsou buď přímo či nepřímo dostupné z vnějšku množiny young a jejich *gc_refs* je nastaveno na GC_REACHABLE. O rozdělení young na dosažitelné a nedosažitelné objekty, se stará metoda *move_unreachable*, která sekvenčně projde celou množinu young, přičemž u jednotlivých objektů testuje jejich *gc_refs*. Pokud jsou objekty dosažitelné z vnějšku young, pak jejich *gc_refs* nastaví na GC_REACHABLE a následně projde a označí jejich potomky, pak vezme sousední objekt. Je-li ten nedostupný, je jeho *gc_refs* příslušně označeno a objekt je vložen na seznam unreachable, pak se zkoumá další objekt. Z toho plyne, že označení objektu za nedostupný není definitivní, protože se může později zjistit, že je daný objekt potomkem jiného z vnějšku dostupného objektu (případně potomek potomka) a být tak označen za dostupný. Celá metoda move_unreachable je vidět v algoritmu 4.2.

```
function move_unreachable(young, unreachable)

    hlavička gc = young.gc.gc_next;
    while nejme na konci young
        hlavička next
        if  gc.gc_refs
            objekt op  =  načti_objekt_z (gc)
            gc.gc_refs  =  GC_REACHABLE
            sleduj_sleduj_dosažitelné_z(op)          //nastaví objekty dosažitelné z op
            next = gc.gc_next                        //vezme sousední objekt

        else

            next = gc.gc_next
            Přidej:na_seznam(gc, unreachable)        //dá gc do unreachable
            gc.gc_refs = GC_TENTATIVELY_UNREACHABLE

        gc = next
```

Alg. 4.2 : Metoda move_unreachable

Po té jsou všechny objekty z young, kde jsou nyní pouze dosažitelné objekty, přesunuty do starší generace. Samozřejmě, že to se netýká případu, kdy je zpracovávána nejstarší generace.

Nyní, když máme od sebe odděleny objekty dostupné a nedostupné, mohlo by se zdát, že je naše práce téměř u konce, nyní by mohlo stačit, když buď cykly mezi objekty v unreachable rozbijeme a odstranění objektů, necháme na makrech čítače referencí nebo dokonce můžeme celý seznam unreachable "zahodit" a místo v paměti tak uvolnit. Jenže všechny objekty, které jsou uloženy na seznamu nelze bezpečně uvolnit z paměti, abychom se mohly nedostupných objektů zbavit, musíme proto nejprve ze seznamu unreachable odstranit ty objekty jejichž uvolnění není bezpečné.

Proto je vytvořen seznam finalizers, který se naplní objekty, kterým se říká finalizers. Seznam unreachable je sekvenčně projit a jednotlivé objekty jsou testovány zda jsou finalizers či ne, pokud ano, jsou přesunuty na stejnojmenný seznam. O to se stará metoda *move_finalizers*.

Ani pak, ale nemůžeme přistoupit k uvolnění objektů. Nebezpečné jsou totiž i objekty, které jsou z finalizers, ať už přímo či nepřímo dosažitelné, proto se i ony musí přemístit na seznam finalizers. To má nestarost metoda *move_finalizer_reachable*, která vysleduje potomky objektů ze seznamu finalizers a pak je přesune na seznam.

Nyní je seznam unreachable prost nebezpečných objektů a může být odstraněn. Ještě než bude zavolána metoda *delet_garbage*, která se o to postará, je nutné se zabavit slabých referencí mezi unreachable a old. Metoda *handle_weakrefs* odstraní reference a zpracuje možná zpětná volání, které mohou nastat. Je nutné dát pozor, aby byly volány jen ty zpětná volání, která souvisejí s referencemi, které pocházejí od živých objektů v old generaci (která je teď vlastně naší zpracovávanou generací), ostatní volání nejen, že není nutné spouštět, ale hlavně to může být nebezpečné, protože by to mohlo vést k znovu oživení mrtvého objektu, který již nemá význam.

Nyní se tedy dostáváme k volání metody *delete_garbage*, která rozbije cykly uvnitř unreachable a odstraní objekty, které jsou v seznamu uloženy. Metoda je vidět na algoritmu 4.3. Je zde vidět, že je seznam sekvenčně projit a u jednotlivých objektů jsou pomocí volání makra clear (viz algoritmus 4.4) rušeny reference na ně ukazující a objekty uvolňovány z paměti pomocí volání dekrementačního makra. Pokud by objekt naše pokusy o jeho odstranění přežil, budeme jeho *gc_refs* nastaveno na GC_REACHABLE a bude následně přesunut do old.

```
function delete_garbage(unreachable, old)

    inquiry clear                                   //Clear musí být typu inquiry

    while není prázdný seznam unreachable
        hlavička gc = unreachable.gc.další          //vezme další prvek seznamu
        objekt op = načti_objekt_z(gc)              // uloží do op ukazatel na objekt z
                                                    // unreachable

        if není tp_clear objektu op NULL
            svyš_Referenci(op)
            clear(op)                               //volá se makro clear
            sniž_Referenci(op)


        if (unreachable.gc.další  == gc)            //objekt je stále dostupný, proto bude
                                                    //přesunut
            přiřaď_objekt_do _generace(gc, old)
            gc.gc_refs = GC_REACHABLE
```

Alg. 4.3 : Metoda delete_garbage

Zbývají nám již jen dvě metody k zavolání. Tou první je *handle_finalizers*, která spojí finalizers s old a vrátí tak tyto problematické objekty zpět tam odkud vzešli, je přitom pravděpodobné, že se s nimi potkáme při dalším zpracování této generace detektorem cyklů. Před tím ale ještě tato metoda pořídí kompletní seznam objektů, které obsahují metody __del__.

Tou druhou metodou je *clear_freelists*, která vrátí operačnímu systému část půjčené paměti tím, že vyprázdní freelisty. Nicméně k tomu nedochází při každém průchodu detektorem, ale pouze je-li kontrolována nejstarší generace.

```
makro Clar(op)
    do
        if op                                   //op je objekt, který chceme uvolnit
            objekt tmp = (objekt)(op)           //vytvoří se kopie objektu
            (op) = NULL                         //původní instance se zruší
            sniž_Referenci (tmp);               //objektu je odebrána jedna
                                                //reference
    while false
```

Alg. 4.4.: Makro Clear

5. Scheme

Jazyk Scheme patří mezi funkcionální dynamicky typované jazyky. Svou syntaxí pak připomíná jazyk Lisp. Díky své jednoduché syntaxi je Scheme využíván k výuce programování či pro vytváření tetovacích programů ve výzkumu, kdy se jeho přehledná a lehce pochopitelná syntaxe hodí.

Scheme má jako programovací jazyk doslova desítky různých implementací. Tato, MzScheme je verze 4.2. a ve svém garbage collectoru využívá implementaci Boemova algoritmu.

5.1. O objektech na haldě

Jazyk Scheme, nebo alespoň tato jeho implementace, která využívá Boehmova algoritmu, disponuje několika předdefinovanými druhy objektů, které lze alokovat na haldu. Některé z nich se chovají normálně tedy mohou obsahovat reference na jiné objekty a mohou být uvolněny garbage collectorem. Jiné nemohou obsahovat ukazatele na jiné objekty a dva druhy jsou neuvolnitelné garbage collectorem. Konkrétně jsou zde tyto typy objektů:

1. PTRFREE – jsou objekty, které ve svých tělech nenesou žádné ukazatele na jiné objekty, ale mohou být uvolněny garbage collectorem.
2. NORMAL – jsou normální objekty, které ve svých tělech mohou obsahovat ukazatele na jiné objekty a mohou bývá uvolněny garbage collectorem.
3. UNCOLLECTABLE - jsou objekty, které mohou obsahovat reference na jiné objekty, ale nemohou být uvolněny garbage collectorem. Jsou jich dva druhy:
   a. AUNCOLLECTABLE – jedná se o atomické objekty (bez referencí na jiné objekty), které nenesou žádnou hodnotu
   b. STUBBORN – objekty neatomické

Objety jsou zde alokovány pomocí dvouúrovňového alokátoru, který rozlišuje mezi velkými a malými objekty. Hranicí mezi velkým a malým objektem je přitom hodnota proměnné *HBLKSIZE*, která je vždy mocninou čísla dvě. Velké objekty jsou pak ty, které mají velikost větší než je polovina *HBLKSIZE*. Jejich velikost je vždy zaokrouhlena nahoru na nejbližší celý násobek *HBLKSIZE* a až pak jsou alokovány pomocí metody *GC_allochblk*.

Malé objekty jsou alokovány v chuncích, které mají rozměr roven *HBLKSIZE.* Každý chunk obsahuje objekty jen jednoho druhu, přičemž všechny musí být stejně velké. Jinými slovy se dá říct, že malé objekty jsou alokovány do těch velkých. Jakmile rozdělíme velký objekt, abychom zde mohli alokovat malý objekt, můžeme zbytek volného místa využít pouze k alokování malých objektů stejné velikosti. Teprve pokud je celý chunk volný, může být umístěn na freelist pro velké objekty dané velikosti.

Tím se dostáváme k freelistům. Scheme využívá freelistových polí. Každý druh objektu má zde vlastní pole freelistů ve všech velikostech. Položky ve freelistech jsou spojeny dohromady pomocí prvního slova v každém objektu.

5.2. Popis činnosti garbagr collectoru

Scheme, podobně jako OCaml, využívá pro nalezení a sběr nedosažitelných objektů algoritmus Mark-and-Sweep. Konkrétně se jedná o Boehmovu implementaci tohoto algoritmu, která nám navíc poskytuje výhody generačního a inkrementačního přístupu, podobně jako možnost paralelního zpracování a podporu vláken.

Collector je nastavitelný, takže se můžeme rozhodnout, zda využijeme všech možností, které nám Boehmův algoritmus poskytuje či zda se například obejdeme bez paralelního zpracování nebo inkrementačního přístupu.

Ke spuštění garbage collectoru dochází v momentě, kdy došlo k nedostatku místa na haldě a zároveň je celkové množství alokací provedených od posledního sběru větší než celková velikost haldy vydělená *GC_free_space_divisor*. Pokud je celkové množství alokací menší, dojde k pokusu o rozšíření haldy.

Popis fáze mark

V každém sběru, collector označí všechny objekty, které by mohly být dostupné z ukazatelů proměnných. Jakmile nemůže říct kde jsou ukazatele proměnných, prohledá následující kořenové segmenty, aby ukazatele našel:

- Registry.
- Zásobník(y).
- Oblast(i) statických dat.

Na začátku fáze mark, jsou všechny kořenové segmenty, které jsem zmínil výše, vloženy na zásobník pomocí metody *GC_push_roots*, aby bylo možné z jejich pomocí nalézt a označit všechny z kořenů dosažitelné objekty.

Marker je strukturován tak, aby umožňoval inkrementační značení. Takže každé zavolání metody *GC_mark_some*, která je zodpovědná za označení dostupných objektů, neprovede označení celé haldy, ale jen malého množství dat. Metoda udržuje stav fáze mark ve formě proměnné *GC_mark_state*, která tak identifikuje jednotlivé pod-fáze. Normální sled pod-fází fáze mark pro sběr pomocí stop-the-world algoritmu (tedy s deaktivovaným inkrementačním přístupem) je následující:

1. MS_INVALID napovídá, že by zde mohl být neoznačený dostupný objekt. V tom případě *GC_objects_are_marked* bude současně false, takže fáze mark pokročil k :
2. MS_PUSH_UNCOLLECTABLE nám značí, že nyní dojde k vložení neuvolnitelnitelných objektů, kořenů na zásobník, a pak označení všeho z nich dostupné. Scan_ptr postupuje přes haldu dokud všechny neuvolnitelné objekty nejsou na zásobníku a objekty dosažitelné z nich nejsou označeny. V tomto bodě, další volání metody *GC_mark_some* zavolá metodu *GC_push_roots* k uložení kořenů na zásobník. To posune fázi mark k :
3. MS_ROOTS_PUSHED nám říká, že jakmile je mark zásobník prázdný, jsou všechny dostupné objekty označeny. Jakmile jsme v tomto stavu, pracujeme pouze na vyprázdnění mark zásobníku. Jakmile je to hotovu stav se změní na :
4. MS_NONE, který značí že dostupné objekty jsou označeny.

Hlavní funkce pro označení objektů *GC_mark_from*, je volána opakovaně několika pod-fázemi, jak se mark zásobník začne naplňovat. Také se opakovaně volá v stavu MS_ROOTS_PUSHED k vyprázdnění mark zásobníku. Samotná funkce je navržena k provádění pouze omezeného počtu označení během každého zavolání, takže může být použita i inkrementačním collectorem. Celá metoda je docela jemně seřízena, ale i tak obvykle spotřebuje velkou většinu času, určeného pro garbage collecting.

Skutečnost že provede jen malé množství práce za zavolání rovněž dovoluje, aby byla použita jako hlavní funkce i při paralelním značení, což jen dokazuje univerzálnost návrhu této metody. V tom případě je tato metoda volána na thread-private mark stacks namísto toho, aby byla volána na global mark stack.

Každý záznam zásobníku mark je zpracován pomocí přezkoumání všech kandidujících ukazatelů v rozsahu, který určí daný záznam. Následně rozhodneme, který z kandidujících ukazatelů je skutečně adresou objektu na haldě. To se učiní v následujících krocích:

1. Kandidující ukazatel je ověřen proti hranicím haldy. Tyto okraje haldy jsou udržovány tak, že všechny aktuální objekty haldy spadají mezi ně.
2. Kandidující ukazatel je rozdělen na dva kusy. Nejvýznamnější bit identifikuje *HBLKSIZE*-velkou stranu v adresovém prostoru a nejméně významný bit určí offset uvnitř oné strany. (hardwarová stránka se ve skutečnosti může skládat z více takových stránek. *HBLKSIZE* je obvykle velikost stránky vydělená mocninou čísla dvě.)
3. Část kandidujícího ukazatele, která obsahuje adresu stránky, je vyhledána v tabulce (ve skutečnosti se jedná o stromovou datovou strukturu). Každý záznam tabulky obsahuje buď nulu, značící, že stránka není součástí haldy, která je zpracovávána garbage collectorem nebo malý integer *n* značící, že stránka je součástí velkého objektu, začínajícího alespoň *n* stran zátky, nebo ukazatel na deskriptor pro stránku. V prvním případě není kandidující ukazatel skutečný ukazatel a může být bezpečně ignorován. Ve zbylých dvou případech můžeme obstarat deskriptor pro stránku obsahující začátek objektu.
4. Začáteční adresa odkazovaného objektu je vypočtena. Stránkový deskriptor obsahuje rozměry objektu(ů) v dané stránce, druh objektu a nezbytné mark bity pro dané objekty. Informace o rozměrech může být využita k namapování kandidujícího ukazatele na počáteční adresu objektu. K urychlení tohoto procesu obsahuje hlavička stránky rovněž ukazatel na předvypočítanou mapu stránkového offsetu k přemístění ze začátku objektu. Použitím této mapy se ve výpočtu počáteční adresy objektu vyhneme možné pomalé operaci se zbytky celého čísla.
5. Mark bit pro cílový objekt je zkontrolován a nastaven. Pokud nebyl objekt předtím označen, je vložen na mark zásobník. Deskriptor je načten ze stránky pro deskriptory.

Na konci mark fáze, jsou mark bity pro zbylé freelisty vyčištěny, pro případ, že by byl free list náhodně označen kvůli zbloudilému ukazateli.

Popis fáze sweep

Již na konci fáze mark jsou velké objekty, které zůstaly neoznačeny okamžitě vráceny na jim určený freelist. Metoda *GC_reclaim_clear* prozkoumá každý chunk, který obsahuje malé objekty, aby se zjistila, zda jsou mark bity u všech zde alokovaných objektů čisté. Pokud je tomu tak, pak je chunk vrácen na freelist velkých objektů. Jestliže ale obsahuje nějaké dosažitelné objekty, pak jsou všechny neoznačené objekty v daném chunku seřazeny do fronty pro pozdější uvolnění z paměti. Pokud se ale rozhodne, že chunk obsahuje příliš málo volného místa, pak již nebude dále prozkoumáván a k uvolnění nedojde.

Samotné malé objekty jsou uvolňovány normálním způsobem a následně umisťovány na freelist, který odpovídá jejich velikosti a druhu. Chunky samotné nemohou být odstraněny dokud je v nich alokovaný alespoň jeden objekt.

Celá sweep fáze se odehrává za použití metod, které jsou umístěny v souboru reclaim.c.

Popis finalization fáze

V teoretické části je zmíněno, že Boehmův algoritmus pracuje ve čtyřech fázích z nichž ta poslední umožňuje volání tzv. finalizačních metod, které mohou být volány některými uvolňovanými objekty.

Objekty, které chtějí zavolat své finalizační funkce jsou již během fáze mark zaregistrovány a pomocí metod *GC_register_disappearing_link* a *GC_register_finalizer* přidány od hash tabilky, která je naalokována vně oblasti zpracovávané collectorem. Požadavky, které jsou v tabulce evidovány jsou zpracovány okamžitě po ukončení fáze mark.

Collector projde celou tabulku a vloží její obsah na mark zásobník. Ten pak označí všechny objekty, které jsou z objektů v tabulce dosažitelné, ale neoznačí objekty z tabulky. Pokud by k označení přece jen došlo, znamenalo by to, že daný objekt je dosažitelný ze sebe sama a je tedy součástí cyklu. Na takový objekt nemohou být volány finalizační funkce, protože by tím mohlo dojít k jeho oživení či případně jiným chybám. Proto budou tyto objekty uvolněny během sweep fáze, aniž by došlo k volání jejich finalizačních funkcí.

Všechny objekty z tabulky, které zůstaly neoznačeny jsou přidány do fronty objektů čekajících na volání finalizéru. Průběh samotného volání je závislý na nastavení collectoru. Buď je možné finalizační funkce provádět implicitně během alokačních volání nebo explicitně jako odpověď na žádost uživatele.

6. Porovnání implementací garbage collectorů

Jednotlivé implementace mají mnoho společných vlastností. Jak koneckonců ukazuje tabulka 6.1. OCaml i Scheme využívají inkrementálního přístupu jako podpory pro Mark-and-Sweep algoritmus. Tam ale kde se OCaml spokojí s obyčejnou implementací Mark-and-Sweep využívá Scheme Boehmův algoritmus, který mu poskytuje podporu paralelního zpracování a vláken, což je ale vykoupeno zvýšenou složitostí výsledné implementace.

Python jde na rozdíl od ostatních dvou jazyků cestou počítání referencí, která mu rovněž zaručuje, že běh programu nebude přerušován garbage collectorem, což je u interpretovaného Pythonu vlastnost velmi žádaná. Na druhou stranu se musí Python vypořádat se zacyklenými objekty, se kterými si počítání referencí neví rady. Zde pak Python využívá svého detektoru cyklů, který cykly na haldě vyhledá a rozbije, takže objekty v nich obsažené pak můžou být odstraněny normálně, pomocí makra které snižuje hodnotu čítač referencí.

	OCaml	Python	Scheme
Počítání referencí	NE	ANO	NE
Mark-and-Sweep	Jen u majoritní haldy	NE	ANO
Stop-and-Copy	Jen u minoritní haldy	NE	NE
Generační přístup	ANO	ANO	ANO
Inkrementální přístup	Jen u majoritní haldy	NE	ANO
Boehmův algoritmus	NE	NE	ANO
Využívá freelist	Jen u majoritní haldy	ANO	ANO
Pole freelistů	NE	ANO	ANO
Setříděný freelist	ANO	NE	NE
Rozlišování slabých a silných referencí	ANO	ANO	ANO
Fragmentace paměti	Jen u majoritní haldy	ANO	ANO

tab. 6.1. Porovnání implementací

Zajímavé je pohled na přístup k alokaci objektů na haldu. Všechny tři implementace využívají freelisty. Python a Scheme pak pro urychlení alokace používají freelistová pole, zatímco OCaml má svůj freelist pro majoritní haldu pečlivě setříděn. Navíc používá OCaml freelist jen pro alokaci na majoritní haldu, zatímco alokace na minoritní haldu zde probíhá mnohem jednodušším způsobem a sice posouváním ukazatele o velikost právě naalokovaného objektu blíže ke konci haldy.

OCaml je také z této trojce jediný. který ve svých generacích používá pro sběr mrtvých objektů různých strategií. Zatímco majoritní halda je spravována algoritmem Mark-and-Sweep, tak minoritní podléhá správě algoritmu Stop-and-Copy, kterému jako to-space slouží právě majoritní halda.

7. Závěr

Tato práce nabízí jedinečné porovnání tří implementací různých garbage collectorů, jejich srovnání a popis jejich činnosti. Každý z těchto collectorů je součástí implementace jiného programovacího jazyka a využívá jiné strategie pro nalezení a uvolnění nedosažitelných objektů z haldy.

Podařilo se mi analyzovat implementace garbage collectorů v implementacích jazyků OCaml, Python a Scheme. Jejich vlastnosti jsme popsal v jednotlivých kapitolách a následně je shrnul v tabulce 6.1. v šesté kapitole.

Jednotlivé implementace mají mnoho společných vlastností. OCaml i Scheme využívají inkrementálního přístupu jako podpory pro Mark-and-Sweep algoritmus. Tam ale kde se OCaml spokojí s obyčejnou implementací Mark-and-Sweep využívá Scheme Boehmův algoritmus, který mu poskytuje podporu paralelního zpracování a vláken, což je ale vykoupeno zvýšenou složitostí výsledné implementace.

Python jde na rozdíl od ostatních dvou jazyků cestou počítání referencí, která mu rovněž zaručuje, že běh programu nebude přerušován garbage collectorem, což je u interpretovaného Pythonu vlastnost velmi žádaná. Na druhou stranu se musí Python vypořádat se zacyklenými objekty, se kterými si počítání referencí neví rady. Zde pak Python využívá svého detektoru cyklů, který cykly na haldě vyhledá a rozbije, takže objekty v nich obsažené pak můžou být odstraněny normálně, pomocí makra které snižuje hodnotu čítač referencí.

OCaml je jediným ze zde popsaných jazyků, který v implementaci svého garbage collectoru využívá pro vyhledání nedostupných objektů více než jednu metodu (pokud u Pythonu nebudeme počítat jeho detektor a je také třeba si uvědomit, že generační a inkrementální přístup sami o sobě mrtvé objekty vyhledat nedokáží). OCaml tak využívá možnosti, kterou mu dává generační přístup a používá v každé ze svých dvou generací jiný algoritmus. Zatímco u majoritní haldy je tom zde již zmíněný Mark-and-Sweep, u minoritní haldy se využívá algoritmu Stop-and-Copy, což omezuje problémy s fragmentací jen na majoritní haldu.

Alokace u všech tří implementací využívá freelistů. Python a Scheme pak využívají freelistová pole, aby alokaci objektů na haldu urychlili, zatímco OCaml udržuje za stejným účelem svůj freelist setříděný.

8. Literatura

1. Appel Andrew W. - Modern Compiler Implementation In C. Cambridge University Press (United Kingdom), 2004. 556 s. ISBN 9780521607650. Kapitola 13, Garbage Collection, s. 273 – 298.

2. Objective Caml [programovací jazyk]. Ver. 3.10.2
URL: <http://caml.inria.fr/download.en.html>

3. Python [programovací jazyk]. Ver. 3.0
URL: <http://www.python.org/download/>

4. MzScheme [programovací jazyk]. Ver. 4.2.
URL: <http://www.plt-scheme.org/software/mzscheme/>